LA-UR-19-26916
Approved for public release; distribution is unlimited.

Title:	C++ Introduction and Best Practices
Author(s):	Lippuner, Jonas
Intended for:	Guest lecture at CU Boulder for Computational Multi-Physics Production Software Development course
Issued:	2019-07-18

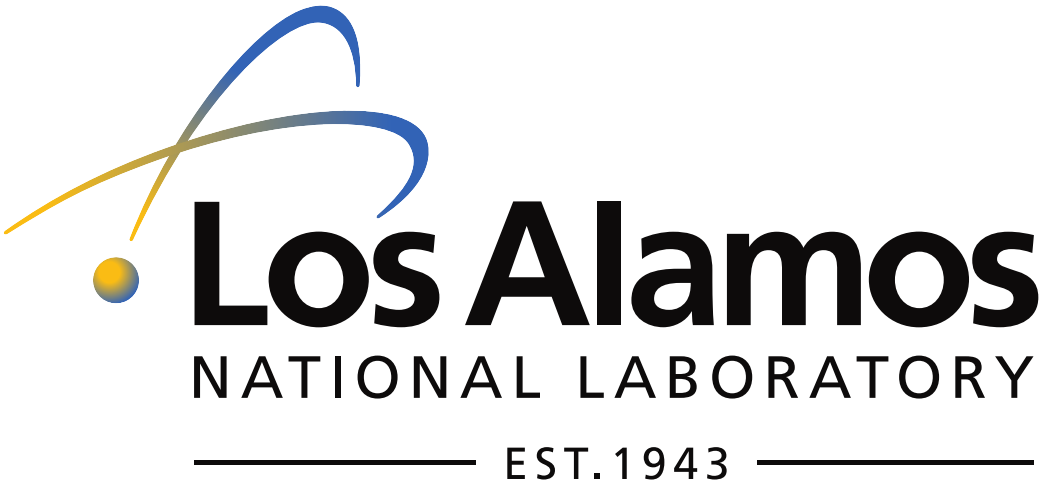
C++ Introduction and Best Practices

Jonas Lippuner

July 11, 2019

Computational Multi-Physics
Production Software Development

CU Boulder
Boulder, CO



Managed by Triad National Security, LLC for the U.S. Department of Energy's NNSA

Resources

- C++ Reference Guides: `http://www.cplusplus.com/` and `https://en.cppreference.com/`
- C++ Core Guidelines: `http://isocpp.github.io/CppCoreGuidelines/`
- C++ Best Practices: `https://github.com/lefticus/cppbestpractices/`
- C++ Coding Standards: 101 Rules, Guidelines, and Best Practices (Herb Sutter & Andrei Alexandrescu. Addison-Wesley, 2004)
- Google C++ Style Guide: `https://google.github.io/styleguide/cppguide.html`
- Tutorials and much more: Google it!

Tools

Bare minimum: text editor + compiler (g++, clang, icpc, etc.)

Highly recommended
(Just a few selected tools I use often, there are many more)

- *Build system:* CMake `http://www.cmake.org/`
- *Version control:* git `https://git-scm.com/`
- *Integrated Developer Environment (IDE):*
  - Visual Studio Code `https://code.visualstudio.com/`
  - Eclipse CDT `https://www.eclipse.org/cdt/`
- *Formatter:* clang-format `https://clang.llvm.org/docs/ClangFormat.html`
- *Documentation:* doxygen `http://www.doxygen.nl/`

C++ crash course

Disclaimer: This is a very brief and incomplete overview of C++ and by no means a comprehensive introduction to C++ programming.

C++ basics

- **Strongly typed:** Each variable needs a type (e.g. `int`, `double`, `std::unordered_map<std::string, float>`)
- **Namespaces:** Types and functions are defined in nested namespaces separated by `::` (e.g. `std::string`)
- **Scope:** Variables live in a scope (inside `{` and `}`) and they get destroyed when they go out of scope
- **Copy by value:** Variables are copied by value by default, use reference (`&`) or pointer (`*`) to avoid copy
- **Const:** Variables, references, pointers can be declared constant (`const`) and they cannot be changed after setting the value
- **Zero-based indexing:** First element has index 0, think of index as memory offset
- **Integer division:** Integer division always truncates fractions (rounding down for positive results): `7/3 = 2, 9/10 = 0, -7/4 = -1`

C++ basics: Control flow

- **Functions:** return type, function name, argument list

```
int sum(const int a, const int b) {
  return a + b;
}
void print(const double val) {
  printf("The value is %.3f\n", val);
}
```

- **If-else:** If, else if, else

```
if (val < 0.0) {
  // do something
} else if (val == 0.0) {
  // do something
} else {
  // do something
}
```

C++ basics: Control flow

- **While loop:** Do something while a condition is true

```
FILE *f = fopen("input.txt", "r");
while (!f.eof()) {
  // read a line from the file
}
```

- **For loop:** for (initialization; condition; increase) statement

```
for (int i = 0; i < 10; ++i) {
  printf("%i\n", i);
}
```

- **Range-based loop:** Since C++11, can loop over elements in a range

```
std::vector<std::string> lines;
for (auto& l : lines) {
  // process line l
}
```

- **Continue:** The `continue` statement skips the rest of the loop body and moves to the next iteration
- **Break:** The `break` statement terminates the loop

C++ basics: Templates

- **Templates** are used to create generic code that can be reused for different types
- **Example:** Generic addition function

```
template <typename T>
T add(const T a, const T b) {
  return a + b;
}
int i = add<int>(3, 8); // or just add(3, 8)
double d = add<double>(-0.3, 1.5); // or just add(-0.3, 1.5)
```

- **STL:** The Standard Template Library contains useful templated algorithms, containers, and functions
  - Dynamic- and static-sized arrays, linked list, deque, stack, set, hash table
  - Sorting, binary search, permutation, search

```
#include <algorithm>
#include <vector>
std::vector<int> numbers { 3, 1, 0, 2, 4 };
std::sort(numbers.begin(), numbers.end());
```

Classes – Basics

- A **class** is an object that can hold data (member variables) and provides functionality (member functions)
- Member variables and functions are **public** or **private** (also **protected**)
- Classes have **constructor** and **destructor** that are executed when a new instance of a class is created or destroyed

```
class Person {
public:
  // constructor
  Person(const std::string name, const int age) :
    name_(name),
    age_(age) {}

  void Print() const {
    printf("%s is %i years old\n", name_.c_str(), age_);
  }
private:
  std::string name_;
  int age_;
};
```

Classes – Inheritance

- A **derived class** is derived from a **base class**, e.g.

```
class Animal {};
class Cat : public Animal {};
class Dog : public Animal {};
```

- **protected** member variables and functions are accessible by derived classes
- **Virtual functions** of the base class can be overridden by the derived class
- **Abstract functions** are functions of a base class that have no implementation (=0), as a result, derived classes are forced to implement them and base class cannot be instantiated

```
class EquationOfState {
public:
  virtual double ComputePressure(const State &state) =0;
};
class IdealGas : public EquationOfState {
public:
  double ComputePressure(const State &state) override { return 1.0; }
};
```

Some select best practices

Mostly following *C++ Coding Standards: 101 Rules, Guidelines, and Best Practices* by Herb Sutter and Andrei Alexandrescu (Addison-Wesley, 2004)

Compile cleanly at high warning levels

- Even though program may be correct with warnings, warnings can reveal bugs and potential issues
- Fix warnings by changing code, not disabling warnings
- Warning-free code makes intent clearer and is thus easier to understand

```
void CreateMesh(const size_t Nx, const size_t Ny) {
  std::vector<double> field(Nx * Nx);
  // ...
}
```

Results in `unused function parameter` warning because `Ny` was not used (by mistake). If parameter truly not needed (e.g. because of inherited function interface, comment it out:

```
void CreateMesh(const size_t Nx, const size_t /* Ny */)
```

Make use of tools

- Use a **version control system** (e.g. Git) to keep track of code changes and to collaborate efficiently
- Use an **automated build system** (e.g. CMake, Make) to build with a single action, or better build automatically when pushing changes to Git repository (e.g. Gitlab, Github, Bitbucket)
- Use an **automated testing system** (e.g. CTest, Google Test) to automatically run unit and regression tests and report failures
- Use a **code formatting tool** to automatically format all the code files (e.g. clang-format) and consider making it a requirement for code updates (like passing tests)

Use a consistent coding style and naming convention

- It is not so important to use a particular coding style, but it is important to use a **consistent** one
- **Distinguish member variables** with a prefix (e.g. `m_`) or a suffix (e.g. `_`); don't use `_` as a prefix (reserved for compiler)
- Common choices:
  - `CamelCase` for types, class names, and function names (except accessors)
  - `snake_case` for variables
  - Accessors are named like the variable
- Prefer `//` for comments rather than `/* */` because latter doesn't nest
- Always use spaces for indentation, never use tabs (configure your editor to indent automatically on tab)

Avoid macros

> *Macros are obnoxious, smelly, sheet-hogging bedfellows for several reasons, most of which are related to the fact that they are a glorified text-substitution facility whose effects are applied during preprocessing, before any C++ syntax and semantic rules can even begin to apply.*
>
> *– Herb Sutter*

```
#define PI 3.14159

double arc = PI * theta;  -->  double arc = 3.14159 * theta;
class Prposal {
private:
  std::string PI;  -->  std::string 3.14159;
};

#define DIVIDE(a, b) a / b
double l = DIVIDE(5.0, r + 1.0);  -->  double l = 5.0 / r + 1.0;
```

Avoid macros

- Blind text substitution
- Ignore scope
- Ignore type system
- Don’t understand C++ semantics (they are from the C days)
- Compiler never sees it, leading to cryptic and hard to debug compiler errors
- Macros are completely divorced from the C++ programming language

Write self-documenting code

- Use **descriptive variable names** avoiding abbreviations, long variable names are ok, using auto-completion, they rarely need to be typed
- Use **const proactively** when declaring variables, functions, and function arguments, especially when using references. This lets the reader know the intention of not changing the variable and it's enforced by the compiler
- Write **meaningful inline comments** but don't wirte comments saying what a line of code does, that should be evident from the code itself
- Use **Doxygen** (or a similar tool) to automatically generate documentation

Separate class implementation from header

- Classes consist of two files: a **header file** (`.hpp`) and a **source file** (`.cpp`)
- Header file declares the interface of the class and it's all that's needed to use the class in some other part of the code (`#include "MyAwesomeClass.hpp"`)
- Header file should be **self-sufficient**, i.e. include all dependent header files
- Enables parallel build of multiple source files
- If implementation changes, only one file needs to be recompiled
- Must use **include guards**

```
#ifndef MY_AWESOME_CLASS_HPP_
#define MY_AWESOME_CLASS_HPP_

class MyAwesomeClass {
  // ...
};

#endif // MY_AWESOME_CLASS_HPP_
```

Make good design choices

- Give **one entity** (class, variable, function) **one cohesive responsibility**
- **Correctness, simplicity, and clarity come first** Complexity opens door for vulnerabilities, bugs, and obfuscation
- **Don't optimize prematurely** It's far easier to make a correct program fast than to make a fast program correct
- **Minimize global and shared data** This reduces coupling between components, increases maintainability and re-usability
- **Hide information** Don't expose internal information and implementation through a public interface, otherwise outside code can manipulate internals, which increases coupling. Should be possible to change internal implementation without other code having to know about it (no ripple effect)

Use proper initialization

- Always initialize variables
- Initialize class member variables in the constructor
- Practice RAII (resource acquisition is initialization), if you use a resource (file, network connection, library), have an object own it. Acquire/initialize the resource in the constructor, release/finalize it in the destructor

```
class LogFile {
public:
  LogFile(const std::string path) :
      file_handle_(fopen(path.c_str(), "w")) {
    // check file was opened
  }
  ~LogFile() {
    fclose(file_handle_);
  }
private:
  FILE * file_handle_;
};
```

Conclusions

- C++ is an incredibly powerful, versatile, and flexible language
- Follow the well-established best practices
- Focus on readability, simplicity, correctness first
- Separation of concerns: write small classes that have a specific purpose, don't try to do everything in one class
- Use consistent style and formatting
- Leverage available tools for automation